Modern Fortran usage and features

Computer techniques for modellers

David Ham

David.Ham@imperial.ac.uk

Imperial College London

The Fortran family

1954 FORMula TRANslator developed at IBM

1961 FORTRAN IV - first portable Fortran

1966 FORTRAN 66 - first standardised Fortran

1977 FORTRAN 77 - huge mass of existing code

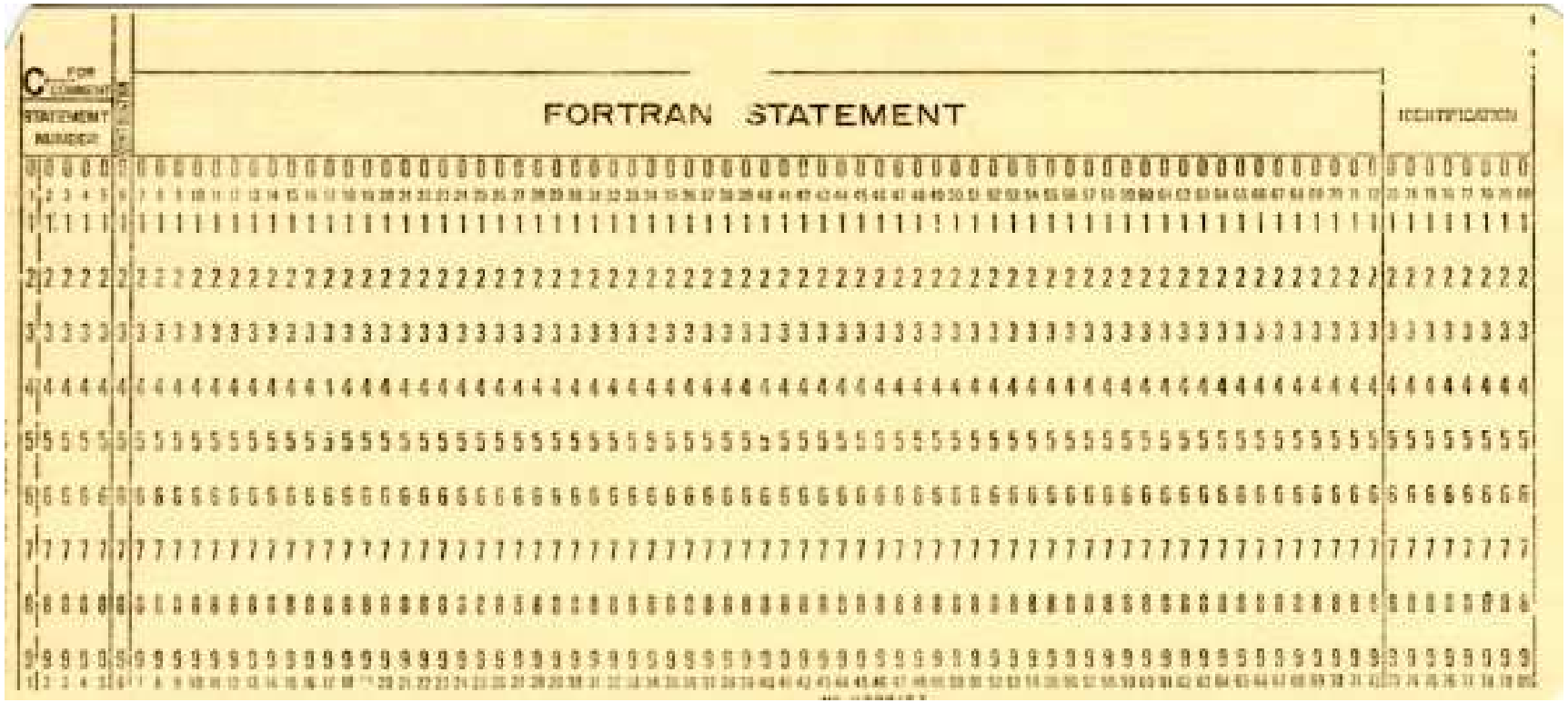
1990 Fortran 90 - major revision extending FORTRAN 77. F90 includes all of FORTRAN 77 plus new features including free form source, dynamic memory allocation and much more.

1995 Fortran 95 - minor tweaks to the standard.

2003 Fortran 2003 - next major revision. Incorporates object oriented programming and proper C interoperability. No compilers yet, though!

Source form

In the beginning there were punch cards:



Source form

From F90 onwards we have a choice of two different ways of writing source code.

```
      program hello_world
C     Fixed form version of the canonical program
        implicit none
        print *,
     &       "Hello World"
C       That was a continued line
      end program
```

```
program hello_world
  ! Free form version of the canonical program
  implicit none
  print *, "Hello &
      &World" ! This is a continued line.
end program hello
```

Source form

- Both free and fixed form source are valid for Fortran90/95. There is no difference to the language features which are valid in each form. .f and .F files are just as much Fortran 90 as .f90 and F90 files.
- Free form source is more readable, less error-prone and make better use of scarce columns. It is to be preferred for all new code.
- In each source form, it is important that all code blocks are indented. Use a Fortran aware text editor to help in the task.

Comments and Names

```
function make_element_shape(loc, dimension, degree, quad, stat) &
      result (shape)
  ! Generate the shape functions for an element. The result is a
  ! suitable element_type.
  !
  type(element_type) :: shape
  ! Loc is the number of vertices of the element, not the number
  ! of nodes!
  ! Dimension is the dimension of the elements (2 or 3)
  ! Degree is the degree of the Lagrange polynomials.
  integer, intent(in) :: loc, dimension, degree
  ! Quadrature information for the element.
  type(quadrature_type), intent(in), target :: quad
  integer, intent(out), optional :: stat
```

Comments and Names

- Names can be up to 31 characters long. Use names which make it immediately obvious what the procedure or variable is for. Local variables can have quite short names (eg i, j, k for loop indices) but procedures and more global names must prioritise description over length.
- Comments should explain everything which won't be instantly obvious to a new person reading the code. Don't repeat what the code does but do explain what it's there for. Often half or more of the lines may be comments.

Modules

```
module mymodule
!My very own module
use some_other_module ! This module is available throughout mymodule.
use yetanothermodule, only:some_name, another_name
implicit none ! This applies to the whole module.

! Module variables go here.

contains ! You only have a contains statement if there are module
         ! procedures
  subroutine my_subroutine(foo, bar)
    ! Module procedures are much like external procedures.

    ! They can also contain internal procedures.

  end subroutine my_subroutine ! "end subroutine" is required.

end module mymodule
```

Compiling Modules

A module must be compiled before any program units which use it. Compilation order can be forced by placing an explicit dependency in the relevant Makefile.in:

```
# This forces Mymodule.o to be made before user.o
user.o: Mymodule.o
```

A leading capital in a module name prevents make confusing a .mod file for a Modula2 source file.

Modules

- Modules provide explicit interfaces for routines. This is crucial as mismatched arguments are one of the most common sources of errors. **Every routine should be in a module**.
- Modules can also aid code structure by encouraging logical grouping and orthogonality.

Argument intent

```
function make_element_shape(loc, dimension, degree, quad, stat) &
      result (shape)
  ! Generate the shape functions for an element. The result is a
  ! suitable element_type.
  !
  type(element_type) :: shape
  ! Loc is the number of vertices of the element, not the number
  ! of nodes!
  ! Dimension is the dimension of the elements (2 or 3)
  ! Degree is the degree of the Lagrange polynomials.
  integer, intent(in) :: loc, dimension, degree
  ! Quadrature information for the element.
  type(quadrature_type), intent(in), target :: quad
  integer, intent(out), optional :: stat
```

Argument intent

Argument intent	Defined on entry?	May be modified?
intent(in)	Yes	No
intent(out)	No	Yes
intent(inout)	Yes	Yes

Argument intent

- Every single dummy argument of every single procedure must have declared intent.
- Declared argument intent is a form of documentation of arguments, improves compiler optimisation and enables automatic error checking of argument use.
- The sole exception to this are arguments with the pointer attribute as these are not permitted to have declared intent in Fortran 95 (this is fixed in Fortran 2003).

Assumed shape arrays

```
function norm(vector)
  ! Find the 2–norm of vector
  real :: norm
  real, intent(in), dimension(:) :: vector
  integer :: i
  norm=0
  do i=1,size(vector)
    norm=norm+vector(i)**2
  end do
  norm=sqrt(norm)
end function norm
```

- Only dummy arguments can be assumed shape!
- The interface must be explicit at the point of call.

Dynamic memory allocation

Automatic arrays simple syntax, usually allocated off the stack. Suitable for small objects.

Allocatable arrays suitable for large blocks of memory. May be made persistent over calls.

Pointers ultimate flexibility. Useful where data needs to be moved around and sorted. Have the potential to cause memory leaks. Pointers are beyond the scope of this talk.

Automatic arrays

The following automatic arrays are declared in the make_element_shape:

```
! Count coordinates of each point
integer, dimension(dimension+1) :: counts
! Derivative of the dependent coordinate with respect to the
! other coordinates:
real, dimension(dimension) :: dl4dl
```

- Automatically created each time the procedure is called and destroyed when it is left.
- Usually start undefined.
- Must not be given the save attribute.

Allocatable arrays

Allocatable arrays are declared using the allocatable elements:

```
! Pressure element numbering map.
integer, dimension(:), allocatable, save :: pgndglno
```

Before use they must be allocated:

```
allocate(pgndglno(totele*mloc))
```

and afterwards, deallocated:

```
deallocate(pgnglno)
```

Variables with the save attribute remain allocated and keep their values even when the routine exits.

Array Operations

Fortran 90 introduces Matlab-style array operations.

```
integer :: a(3,2), b(3,2), x(3), y(3), z(2)

a=0 ! Set every element of a to 0

a=a+b ! To each element of a add the corresponding vector of b.

a(1:2,1)=z ! Colons can be used to select sub–arrays.

a(:2,1)=z ! Leaving off a limit on a colon implies that
          ! the maximum extent in that direction is chosen.

y=b(:,1)*x ! A bare colon means the whole of that dimension.

x=y+z ! This is not allowed because the dimensions don't match.

x=y+2 ! A scalar will be repeated to match any size array.
```

Arrays and Procedures

Fortran 90 provides a wide range of procedures which enable array and matrix operations:

```
real :: a(3,2), b(3,2), c(2,2), x(3), y(3), z(2), e
x=(/1,2,3/)   ! Vector literal constants are written with (/ /).
a=reshape((/1.5, .7, .3, .4, .6, .8/),shape(a))

c=matmul(transpose(a), b) ! Matrix multiply a' * b.
z=matmul(x,a) ! Matrix multiply x' * b
e=matmul(a,(/1.2,1.5/)) ! Matrix multiply a by the vector given.

e=dot_product(z,b(2,:))

print '(2f15.8)', transpose(a) ! To print an array in matrix
                               ! order, print its transpose.

e=size(a)     ! e is 6, the number of entries in a.
z=shape(a)    ! z is (/3,2/).
```

Vector Subscripts

A rank 1 array can be used as a subscript to a vector:

```
real :: a(20)
integer :: v(5)=(/1, 7, 5, 9, 16/)
integer :: w(5)=(/1, 7, 5, 9, 5/)
integer(i)

forall (i=1:20)
  a(i)= 0.5*i
end forall

print *, a(v) ! prints 0.5 3.5 2.5 4.5 8

a(v)=2.0*a(w) ! This is allowed.

a(w)=2.0*a(v) ! Many–to–one assignments are not allowed.
```

A warning for Matlab users!

Fortran's array syntax is similar, but not identical to, Matlabs. In particular, strides are handled differently. In Matlab:

```
% This returns the odd elements of x
x(1:2:length(x))
```

while in Fortran the same expression is written:

```
! This returns the odd elements of x
print *, x(::2)

! This is equivalent
print *, x(1:size(x):2)
```

Vector assignment features

Two special constructs allow for more complex assignments. The where statement allows an assignment depending on parts of an array:

```
where (a<0.0)
  a=0.0 ! Zero all negative entries in a
end where
```

Forall allows the assignment to depend on the location in the array:

```
forall (i=1:20)
  a(i)= 0.5*i
end forall
```

Some more array functions

```
logical, dimension(10,5) :: l
real, dimension(10,5) :: a

print *, any(a==0) ! .true. if any element of a is 0.

print *, all(a==0) ! .true. if all elements of a is 0.

print *, count(l)  ! Number of true entries of l.

print *, maxval(a,2) ! Vector of maximum row elements in a.

print *, minval(a) ! Minumum entry in a

print *, product(a(1:3,:)) ! Product of first 3 rows of a.

print *, sum(a,1) ! Vector of column sums of a.
```

Array hints

- Judicious use of array operations can radically shorten code and improve optimisation. Array operations are often more readable than multiple loops.
- An important speed rule is to remember that the first index should vary fastest. Fortran arrays are column major. This is the opposite of C.
- Zero sized vectors are allowed in all contexts and work naturally.

Modern loop constructs

```
integer, dimension(4, totele) :: neigh ! List of element neighbours
integer :: i,ele
logical :: sorted
sorted=.false.

outerloop: do ! A labelled do loop.
   ele_loop: do ele=1,totele
      do i = 1, 4 ! Loop over neigbours
         ! Jump to the next element when the neighbours are done.
         if (neigh(j,ele)==0) cycle ele_loop

         ! Some code for sorting elements

         ! Check if we are done
         if (sorted) exit outerloop
      end do
   end do ele_loop
end do outerloop
```

Select case

```
! The hundreds digit of geobal controls the sort of discretisation.
select case (mod(abs(geobal)/100,10))
case (0)
    fe_method=GEO_CG
case(1)
    fe_method=GEO_DG
case default
    ! ALWAYS test the default case!
    ERROR('unknown discretisation')
end select
```

- Select can also be used with character variables.
- You can select for ranges ('a':'c') or lists (1,3,5:7)